Cody context architecture

Sourcegraph Whitepaper – June, 2023

Overview

Cody is an AI coding assistant that lives in the editor and the web UI. Cody can find, explain, and write code, in any major programming language. Using Large Language Models (LLMs) and the code graph, Cody provides context-aware answers that help devs write features and fix bugs faster. An overview of Cody's architecture can be found here.

Context awareness is key to providing the highest quality responses to users. A metaphor we have used is that a raw LLM is like a booksmart programmer who has read all the manuals but doesn't know a company's codebase. By providing context from a company's codebase along with the LLM prompt, the LLM can generate an answer that is relevant to that codebase.

The key value proposition of Cody is that if Cody has the best, most relevant context about a company's codebase then it will be able to provide the best answers. Cody will be able to:

- Answer questions about that codebase
- Generate code that uses the libraries and style of that codebase
- Generate idiomatic tests and documentation
- And, in general, reduce the work developers need to do to go from the answer provided by the LLM to delivering value in their organization

This document goes into detail on how we provide this context to an LLM.

How Cody generates a prompt

A large language model (LLM) is a type of machine learning model which takes a natural language prompt and can generate text in response to the prompt. The LLM does not know specifically about a user's code or what the user is trying to achieve.

This is where Cody comes in. When a user queries Cody, Cody generates a prompt that is specifically designed to help answer questions about the user's code.

The prompt can be divided into three parts:

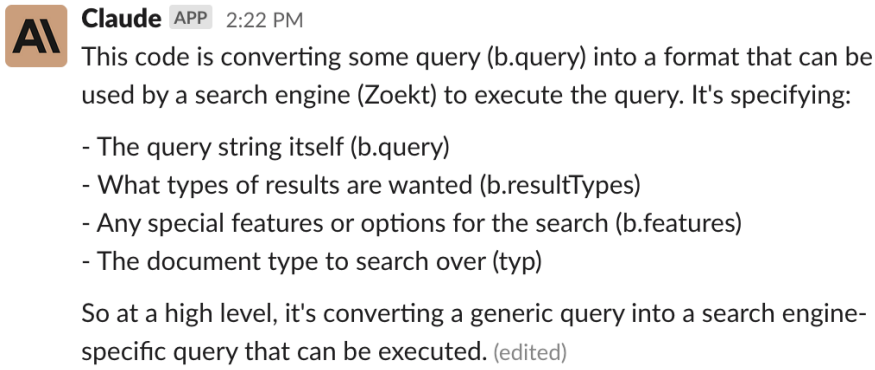
- Prefix: an optional description of the desired output; these come from predefined "recipes" which define tasks the LLM can perform
- User input: the information provided by the user
- Context: Additional information that helps the LLM provide a relevant answer

For example, after the user clicked on the "Explain high-level" recipe, the resulting prompt might look like this (the labels on the left are not part of the actual prompt):

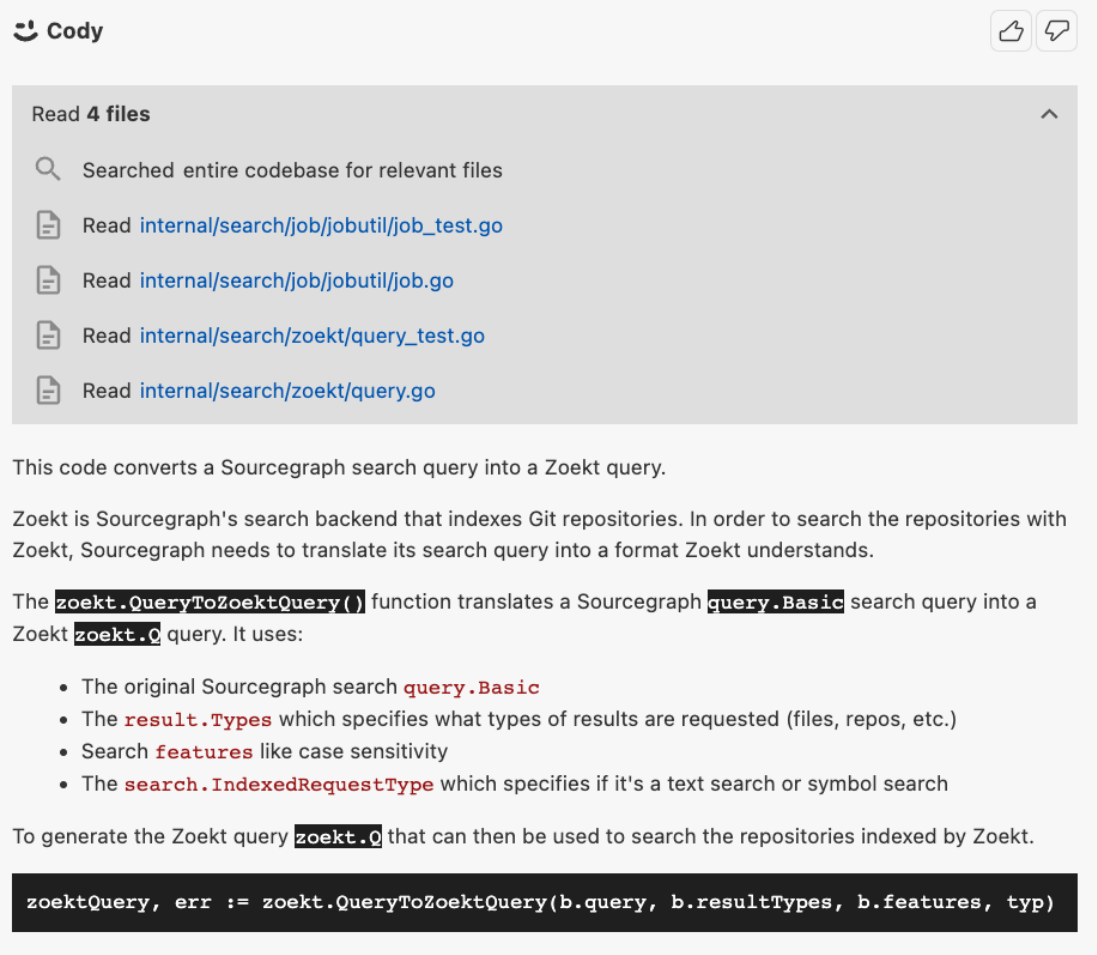
Recipe Prefix	`Explain the following Go code at a high level. Only include details that are essential to an overall understanding of what's happening in the code.`
User Input	`zoekt.QueryToZoektQuery(b.query, b.resultTypes, b.features, typ)`
Context	`[Contents of sourcegraph/sourcegraph/internal/search/zoekt/query.go]`

A prompt like this is sent to the LLM. The information contained in the prompt is the only information the LLM has beyond its baseline model. Because the LLM lacks information about a user's code base, context makes a big difference.

We can illustrate this with an example. If we send the prompt above to Claude *without* the context, we get the following result:



When we send the same prompt to Cody, we include context from multiple files, including `query.go`,which implements `QueryToZoektQuery`. The answer we get is much more specific to Sourcegraph. It notes that Zoekt is Sourcegraph's search backend, gives more background context, and provides much more specific details about the code. The LLM underneath is still Claude. The difference comes from providing the context.



How Cody selects the right context today

Where does context come from? How do we make sure it is good?

The quality of the context is critical, even with increasing context window sizes[1]. For one, with Enterprise code bases continuing to grow[2], even huge context windows are not enough to send everything. Even in cases where sending everything is feasible, sending excess context can increase query cost, data egress, and response time. The costs of not being selective can add up. Effective context selection will continue to be an important part of interacting with LLMs.

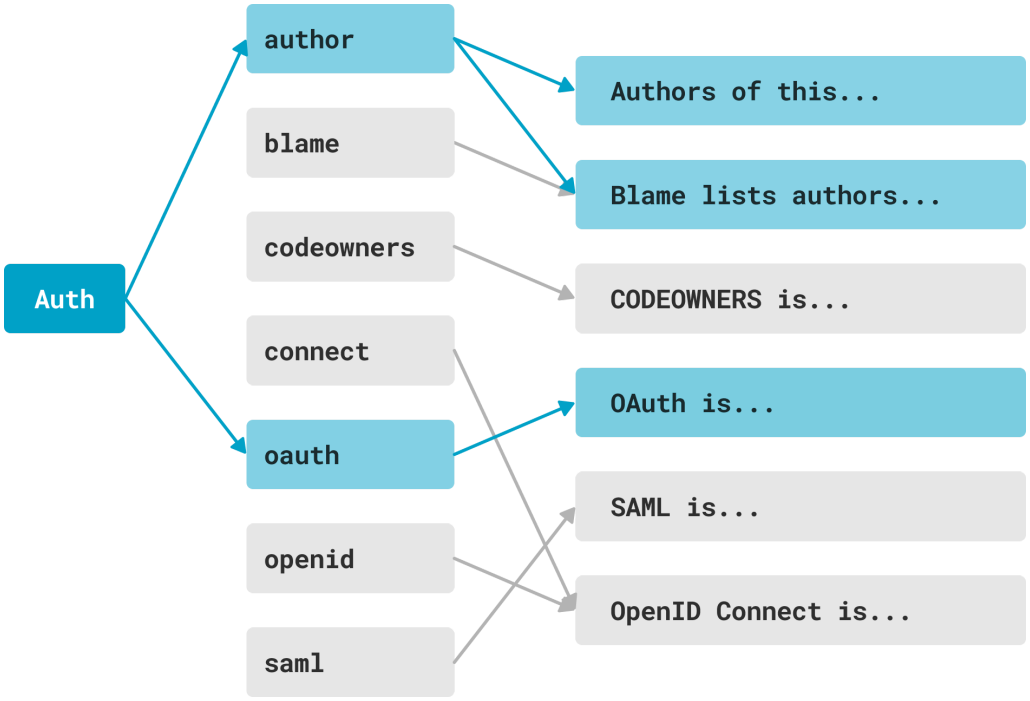
Cody gets context by searching for relevant code snippets. It does this in one of two ways: Keyword Search and Embeddings.

[1] Such as Anthropic's recent introduction of a 100k context window
[2] See Sourcegraph's report on Big Code in the AI Era

Keyword search

Keyword search is the traditional approach to text search. It splits content into terms and builds a mapping from terms to documents. At query time, it extracts terms from the query and uses the mapping to retrieve candidate documents. This is how Google search traditionally worked, so it's pretty powerful.

In the simple non-code example below, the query "Auth" would match both "OAuth" and "Author" (assuming substrings are supported). It would not match the related statements "SAML" and "OpenID Connect" which are also common authentication or authorization methods.



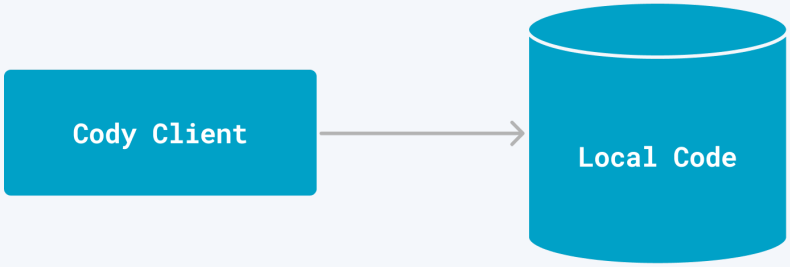
A simple example of keyword search

When using a codebase that has not had embeddings generated (see below), Cody may use keyword search as a fallback, depending on the particular client or extension in use.

Keyword Search Implementation

Want more details? Here's how the Cody VS Code extension uses local keyword search works when embeddings are not available.

1. Extract stemmed[3] keywords and remove stop words[4].
2. Run a tool (ripgrep) over the files in the workspace to look for files that contain exactly those keywords
    - We exclude filesthat are not expected to add relevant context, such as files listed in .gitignore and binary files
3. Score and rank the results using standard search Term Frequency and Inverse Document Frequency
4. Add the files with the highest scores as context



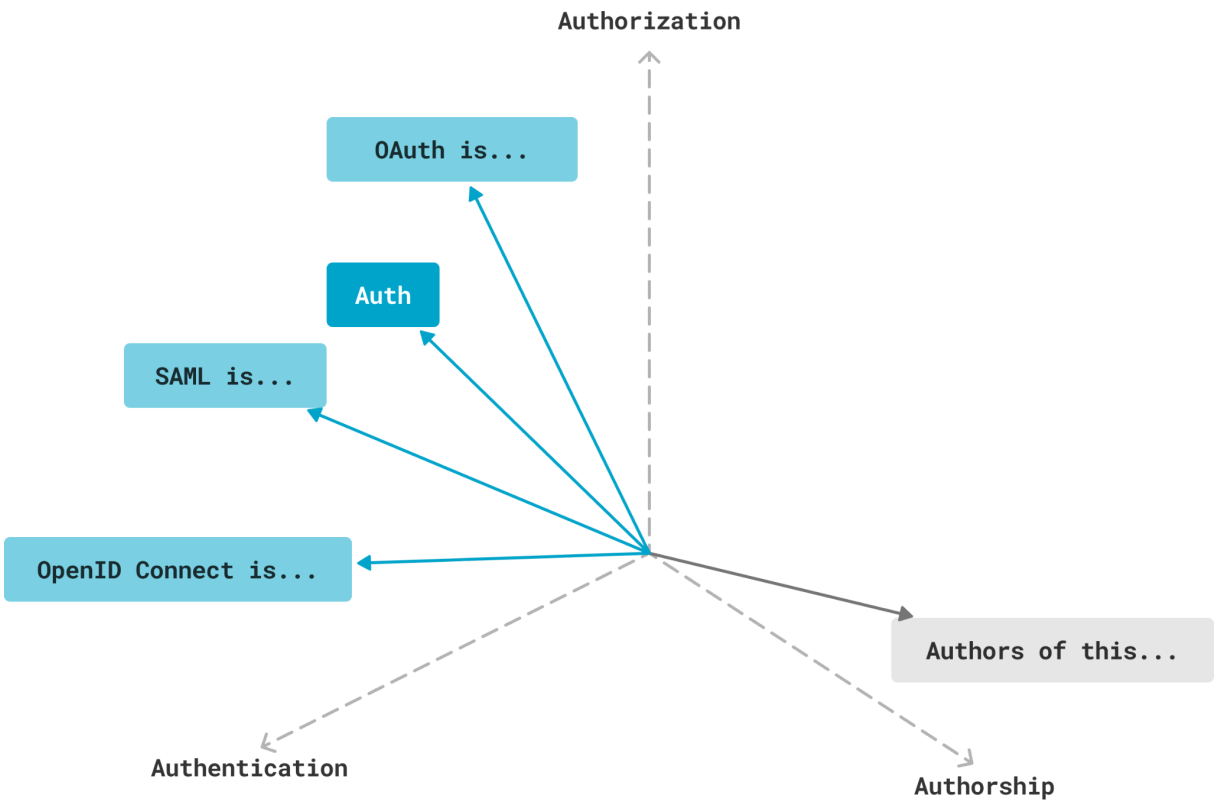
Embedding search

Keyword search matches documents based on how many words they share with the query (perhaps using synonyms or bigrams such as “New York”). However, the most relevant document may have few words in common. We want to look at the meaning of words in context, not just their textual form.

We use a technique the NLP community developed, text embedding. Embeddings encode words and sentences as numeric vectors. These vector representations are designed to capture linguistic content of text, and can be used to measure the similarity between a query and a document based on meaning.

This is especially relevant for code. Code often uses abbreviations or library names to refer to the same concept by different names. For example, using embeddings “Auth” would match “OAuth”, “OpenID Connect”, and “SAML”, which are all common authentication or authorization methods. They should not match “Authorship” since it is not commonly abbreviated as “Auth”.

[3] Stemming is the process of mapping a word to it’s linguistic step. E.g., “days” to “day”
[4] Stop words are common words which are filtered out to improve relevance.



A simple example of embedding space.

Embeddings are the preferred method for fetching context. Compared to keyword search, embeddings search:

- uses all of the code in the specified repositories (not just code in the local workspace)
- matches code based on the meaning of the query (not the exact terms)

Embeddings Search Implementation

Want more details? Here's how embeddings search works under the hood.

1. Cody sends the query to the configured Sourcegraph instance, specifying the repos to search over.
2. The Embeddings Serving subsystem finds the nearest match by:
    a. using an external Embeddings Generation service to convert the user query into an embedding vector
    b. searching over the embeddings stored in a Vector Database for the specified repositories for the ones that

are most similar to the query
3. The Embeddings Serving subsystem returns the matching code chunks along with information about the files those chunks came from

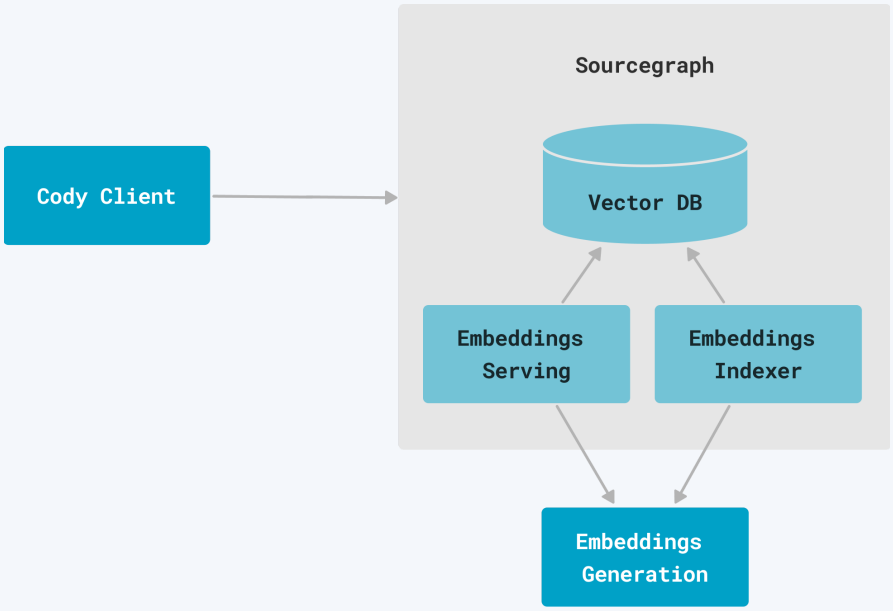
The Embeddings Serving subsystem is a custom backend that does brute force search. It currently scales to around 200 million lines of code.

Embeddings search depends on the relevant repositories having already been converted into embeddings by a batch job.

1. An administrator configures the repositories to be indexed.
2. The Embeddings Indexer requests all files in those repos.
3. Each file is divided into chunks.
    a. We exclude files are not expected to add relevant context, such as files listed in .gitignore and binary files
4. The Embeddings Indexer calls the Embeddings Generation Service for each chunk.
    a. Chunks are embedded using the same embedding model that will be used for search.
5. The embeddings are loaded into the Vector Database associated with the Sourcegraph instance.

This process is repeated periodically to create or update embeddings as code changes. This improves freshness and reduces cost relative to reindexing all the code for any change.

Currently, we use the OpenAI as the service for generating embeddings.



Embeddings vs keyword search

Are embeddings really better than keyword search? Since Cody with embeddings searches over whole repos while Cody with keyword search only searches over the local VS Code workspace, it's fair to ask whether Cody with embeddings is better because of the *embeddings* or just because it looks at more code.

We ran an evaluation of embeddings against keyword search using the CodeSearchNet data. Looking at Normalized Discounted Cumulative Gain (NDCG) over the first 20 returned results, we compared the performance of:

- Embeddings models:
  - OpenAI
  - all-mpnet-base-v2
- Keyword search
  - ripgrep
  - Elasticsearch

Using OpenAI embeddings as the baseline, we saw the following relative quality:

ripgrep	Elasticsearch	OpenAI	all-mpnet-base-v2
77%	95%	100%	119%

Although a true keyword search engine was able to nearly match the OpenAI embeddings model, the result from all-mpnet-base-v2 shows that the right embeddings model can outperform keyword search. Any embeddings model is much better than ripgrep[5].

That said, it doesn't need to be an "either/or". Different search methods can be blended, for example, combining embeddings with keyword search when exact matches are useful.

Improving context

We are continually working to improve context search. Since Cody's initial release, we have already added support for searching over multiple repositories, seamless updating of embeddings, and scaled the current vector database by 10x. Here's where we are going from here.

[5] The advantage of ripgrep is that it is easy to run locally and on demand.

Better embeddings

Today Sourcegraph uses the OpenAII Embeddings API. Our use of this API is stateless. Sourcegraph calls the OpenAI Embeddings API, generates embeddings for repositories based on the user or company's configuration, and then stores them in the vector database associated with the relevant Sourcegraph instance.

This model has been effective for us so far, but it presents some challenges.

Using OpenAI requires that we send a customer's whole code base to a third party service provider. Although we have a 0-retention policy for both LLM and embeddings providers, some customers prefer not to send their entire code base to an entity they do not have a direct relationship with.

Another challenge is that OpenAI embeddings are large. Each embedding is represented by 1536 floating point numbers, which adds up to a lot of RAM. Using smaller embedding vectors would allow the Embeddings Serving subsystem to scale to more code with the same amount of RAM. It also allows records to be scanned more quickly which makes searches faster.

As the note comparing keyword search and embeddings shows, another challenge with OpenAI embeddings is that other embeddings models perform better — even though they're smaller.

To solve these problems, we are replacing OpenAI embeddings with a Sourcegraph managed Embeddings API. This API would be a stateless API that is managed by Sourcegraph[6]. It would generate embeddings directly; it would not call out to another service[7]. The managed service would have a 0-retention policy, with the generated embeddings continuing to be stored in the vector database in a customer's Cloud managed or self-hosted Sourcegraph instance.

Results so far are promising. Even off the shelf, we're seeing smaller models perform up to 20% better on our tests than the OpenAI embeddings. (Models we've evaluated include AllDistilRobertaV1, AllMPNetBaseV2, E5, AllMiniLML6V2, and SentenceT5.)

Beyond code

Developers answer questions with more than just code. Sometimes the best answer to a user's query is found in a company's knowledge base or ticket system. We plan to add support for these and other data providers. Because embeddings encode meaning, we believe they are particularly well suited to finding commonalities across disparate data sources such as code, documentation, bug reports, logs, and more.

[6] We do not currently plan to provide self-hosted functionality for generating embeddings.
[7] We will temporarily retain the option to use OpenAI for generating embeddings to aid in the transition to a new model.

Scaling embeddings

Code bases are getting larger and larger. To handle the needs of large and growing code bases, we need to be able to handle even larger amounts of code while still quickly providing the best context. We plan to offer alternatives to our custom built embeddings server to handle even larger code bases. Systems that support more efficient vector search, on-disk data storage, and horizontal scalability can scale to ever larger datasets. However, these systems also come with deployment challenges. We want to make sure our customers have a choice: an easier to deploy model that works well on moderate code bases or a more complex solution that can scale to even the largest code bases.

Deeper code graph integration

Today, Cody utilizes the power of the code graph to be able to create embeddings for a customer's whole code base. However, we are still treating code largely as text. We are working to push the power of the code graph even further by using the structure of code to allow better context fetching. For example, we can take code structure into account when we chunk code for embeddings. We can use the call graph to fetch context that may not be textually related but is related through usage. For example, neither embeddings on code text nor keywords are good for answering queries of the type "Find all call paths between A and B". The code graph makes following these relationships easy.

Better keyword search

Embeddings give higher quality results than keyword search. However, keyword search is still a valuable fallback. Embeddings search requires preprocessing the code base, which can take awhile and may not be worthwhile for all repositories, e.g., rarely used repositories. We are working on utilizing our existing Code Search infrastructure to provide better keyword search in cases where embeddings are not available.